Harnessing Curiosity to Increase Correctness in End-User Programming

A. Wilson, M. Burnett, L. Beckwith, O. Granatir,
L. Casburn, C. Cook, M. Durham, G. Rothermel
Dept. of Computer Science, Oregon State University, Corvallis, OR 97331

Technical Report 02-60-08
Sept. 23, 2002

ABSTRACT
Despite their ability to help with program correctness, assertions have been notoriously unpopular—even with professional programmers. End-user programmers seem even less likely to appreciate the value of assertions; yet end-user programs suffer from serious correctness problems that assertions could help detect. This leads to the following question: can end users be enticed to enter assertions? To investigate this question, we have devised a curiosity-centered approach to eliciting assertions from end users, built on a surprise-explain-reward strategy. Our follow-up work with end-user participants shows that the approach is effective in encouraging end users to enter assertions that help them find errors.

Keywords
Assertions, end-user software engineering, curiosity

INTRODUCTION

There has been considerable work in empowering end users to be able to write their own programs, and as a result, end users are indeed doing so. In fact, the number of end-user programmers is expected to reach 55 million by 2005 in the U.S. alone [3], writing programs using such devices as special-purpose scripting languages, multimedia and web authoring languages, and spreadsheets. Unfortunately, evidence from the spreadsheet paradigm, the most widely used of the end-user programming languages, abounds that end-user programmers are extremely prone to errors [18]. This problem is serious, because although some end users' programs are simply explorations and scratch pad calculations, others can be quite important to their personal or business livelihood, such as for calculating income taxes, e-commerce web pages, and financial forecasting.

We would like to help reduce the error rate in the end-user programs that are important to the user. Although classical software engineering methodologies are not a panacea, there are several that are known to help reduce programming errors, and it would be useful to incorporate some of those successes in end-user programming. Toward this end, we have been working on a vision we call "end-user software engineering," a holistic approach to the facets of software development in which end users engage. Its goal is to bring some of the gains from the software engineering community to end-user programming environments, *without* requiring training or even interest in traditional software engineering techniques. One of the features in end-user software engineering is assertions.

The idea of assertions for end users may on the surface seem counterproductive. It is well known that, despite assertions' benefits in detecting run-time errors [20], even professional software developers are often unwilling to use assertions. However, empirical data [4] show that end users can effectively use assertions of the type presented in this paper—if the correct assertions have already been provided to them—and that doing so leads to significant improvements in debugging correctness and debugging efficiency. Of course, these strong benefits from assertions can be realized only if users can be enticed into entering assertions and acting upon them.

This paper addresses that problem: how to entice users to enter assertions, and further how to use assertions to encourage the user to think about the portions of the program that are likely to be erroneous. The essence of our strategy is to first arouse users' curiosity through the element of surprise, and to then encourage them, through explanations and rewards, to follow through with assertions. We will refer to this strategy as our surprise-explain-reward strategy.

Research in curiosity indicates that surprising by violating the user's assumptions can trigger a search for an explanation [9]. The violation of assumptions indicates to the user the presence of something they do not understand. According to the *information-gap* perspective [15], a revealed gap in the user's knowledge focuses the user's attention on the gap and leads to curiosity, which motivates the user to explore in order to close the information gap.

In this paper, we present our surprise-explain-reward strategy to provoke the user's curiosity about assertions and about the program's correctness. We show the strategy's instantiation in our end-user software engineering environment for the spreadsheet paradigm, and we then use this environment to investigate the user behaviors this strategy encouraged and the results of their behaviors.

RELATED WORK

Other than the body of research about visually depicting relationships among program objects, there has been little research aimed at helping reduce errors in end-user programming. One notable exception is the work on outlier finding, which points out potential errors in text documents by using statistical outlier algorithms and then highlighting the outliers to the users [16].

On the other hand, there is a wealth of research on the subject of interactive explanations. The most relevant to our work has been done in the constraint programming paradigm on explanations particularly for debugging. Jussien and his colleagues have devised explanations to help both programmers and the end users running the programs to spot the causes for a constraint program failing [10]. Our explanations, while relevant to debugging, have as their primary purpose encouraging users to *use* debugging features provided by the system.

DiGiano et al. have suggested building learning supports into programming systems [6]. They list five facilities with which to do this, including integrating learning opportunities into normal programming activity by using annotations. One example is the character "Marty," the annotated assistant of ToonTalk. A difference between that approach and ours is that, whereas Marty provides help relating to correct syntactic use of programming constructs, our explanations are primarily about helping with semantics of the objects in the environment.

Expanding on the traditional notion of help, Lieberman described interfaces that give and take advice [13]. That is, the computer gives the user advice, and the user can give advice back to the computer. Our approach also offers advice, but only takes advice in a few limited ways. Also, unlike the taking of advice Lieberman discusses, our system does not learn from the actions of the user.

Many end-user programming languages (e.g., [14]) provide immediate visual feedback about semantics, and this can be characterized as the primary use of "reward" in end-user programming to date. One of the cognitive dimensions [8], a framework for providing a common vocabulary to visual language designers, describes this feature as "progressive evaluation." Novices and experts alike rely on evaluating their problem-solving progress often, which progressive evaluation provides. An example of a programming environment that supports progressive evaluation is spreadsheets. For example, making a change to a formula immediately changes other cells that depend on the changed cell. According to Nardi, progressive evaluation is one feature that makes spreadsheets accessible to end users [17].

CONNECTING CURIOSITY WITH ASSERTIONS

Research about curiosity (surveyed in [15]), points out that if an information gap is illustrated to the user, the user's curiosity about the subject of the illustrated gap may increase, potentially causing them to search for an explanation. Without challenging their assumptions and arousing their curiosity, as the information-gap perspective explains and much empirical programming literature bears out [18, 24, 12], users are likely to assume that their programs are more correct than is warranted. This is the motivation for the first component of our surprise-explain-reward strategy: to arouse users' curiosity, through surprise, enough that they search for explanations. Thus, the first component of our strategy might be characterized as following a "principle of *most* astonishment."

The strategy is used in two situations: first to entice users to use the features, and later to help them benefit from the features. In the first situation, the strategy attempts to surprise the user into entering assertions by suggesting assertions that are editable as part of a "help me test" device the user has chosen to invoke. If the user becomes curious, she can find out more via an explanation system whose viewport is tool tips. If the strategy causes the user to act, by acknowledging a suggested assertion as being correct or by editing a suggested assertion, the second situation becomes possible. This time, the surprise comes when an assertion identifies a potential bug. The users can again look to explanations to explain the surprise and suggest possible actions. If the user successfully fixes the bug identified by the assertion, they see that the program's behavior is more correct, a clear reward for using assertions.

The Setting's Opportunities for Curiosity

This work is part of a highly integrated and incremental concept of software engineering support for end users. It is this setting that provides the opportunities for arousing curiosity and following through on it with the surprise-explain-reward strategy.

A continually evolving prototype of the end-user software engineering concept exists for Forms/3 [5], a member of the spreadsheet paradigm. One of its components is the "What You See Is What You Test" (WYSIWYT) methodology for testing [22]. WYSIWYT allows users to incrementally edit, test and debug their formulas as their programs evolve, visually calling users' attention to them by painting their cell borders red (light gray in this paper). Tested cells are painted blue (black), and partial testedness is depicted in purples (grays) along the continuum from red to blue (light gray to black); see Figure 1. Whenever the user notices that a cell's value is correct, he or she checks it off in the checkbox in its corner, increasing the testedness. Empirical work has shown that the WYSIWYT methodology is helpful to users [12] but, even with additional visual devices such as colored arrows between formula subexpressions to indicate the relationships remaining to be covered, after doing a certain amount of testing, users sometimes find it difficult to think of suitable test values that will cover the as-yet-untested relationships. At this point, they can invoke a "Help Me Test" feature. This feature provides an opportunity for surprise.

The "Help Me Test" (HMT) feature [7] suggests test values for user-selected cells or user-selected dataflow arrows. HMT then tries to find inputs that will lead to coverage of an untested portion of the user's selections, or of any cell in the program if the user does not have any cells or arrows selected. HMT cannot always find a suitable test case, but in performance experiments it has succeeded in less than 4 seconds approximately 90% of the time [7]. There is also a Stop button available, if HMT is deemed as taking too long.

HMT's efforts to find suitable test values are somewhat transparent to the user—that is, they can see the values it is considering spinning by. The transparency of its behavior contributes to the understandability of the surprises we have added to it, and it also provides an opportunity for rewards, as we will explain later.

Assertions

As mentioned in the introduction, there is an assertion feature [4] in the environment. (Our system terms these "guards" when communicating with users, so named because they guard the correctness of the cells.) Assertions protect cells from "bad" values, i.e., from values that disagree with the assertion(s). Whenever a user enters an assertion (a *user-entered assertion*) it is propagated through formulas creating *computer-generated assertions* on downstream cells. The user can use tabs (not shown) to pop up the assertions, as has been done on all cells in Figure 1. The stick figure icons on cells Monday, Tuesday, … identify the user-entered assertions. The computer icon, on cell WDay_Hrs, identifies a computer-generated assertion, which the system generated by propagating the assertions from Monday, Tuesday, …, through WDay_Hrs's formula. A cell with both a computer-generated and user-entered assertion is in a conflict state (has an *assertion conflict*) if the two assertions do not match exactly. The system communicates an assertion conflict by circling the conflicting assertions in red. In Figure 1 the conflict on WDay_Hrs is due to an error in the formula (there is an extra Tuesday). Since the cell's value in WDay_Hrs is inconsistent with the assertions on that cell (termed a *value violation*), the value is also circled.

Users have two concrete syntaxes for entering assertions onto a cell: one textual, and one primarily graphical. Examples of the textual syntax are in Figure 1. Or-assertions are represented with comma separators on the same line (not shown), while and-assertions are represented as multiple assertions stacked up on the same cell, as with cell WDay_Hrs. (And-assertions must always agree.) It is possible to omit either endpoint from a range, allowing for relationships such as <, <=, and so on. Further information on the relationships supported, the assertions' relative power, and the graphical syntax can be found in [4]. But, will users ever choose to enter assertions?

Blackwell's model of attention investment [2] is one model of user problem-solving behavior that suggests users will not want to enter assertions as we have just described them. The model considers the costs, benefits, and risks users weigh in deciding how to complete a task. For example, if the ultimate goal is to forecast a budget using a spreadsheet, then exploring an unknown feature such as assertions has cost, benefit, and risk. The cost is figuring out what assertions do and how to succeed at them. The benefit of finding errors may not be clear until long *after* the user proceeds in this direction. The risk is that going down this path will be a waste of time or, worse, will leave the spreadsheet in a state from which it is hard to recover. What the model of attention investment implies is that it is necessary not only for our strategy to make the users curious about assertions, but also to make the benefits of using assertions clear from the outset. This is what our surprise-explain-reward strategy sets out to do.

REALIZING SURPRISE-EXPLAIN-REWARD: HMT ASSERTIONS

The first step of our strategy is to generate a meaningful surprise for the user. That is, the system needs to violate the user's assumptions about their program. We have devised a pseudo-assertion for this purpose, termed an *HMT assertion* because it is produced by HMT. An HMT assertion is a guess at a possible assertion for a particular cell.

Surprises: Garnering User Attention

HMT assertions exist to surprise and thereby to create curiosity. The guesses are reflections of HMT's behavior. That is, they report the range of HMT's attempts to find suitable test cases. For example, in Figure 2, the HMT assertion for cell Monday is "-1 to 0." This indicates that HMT has considered values for Monday between -1 and 0 before it settled upon its current value of -1. If HMT is

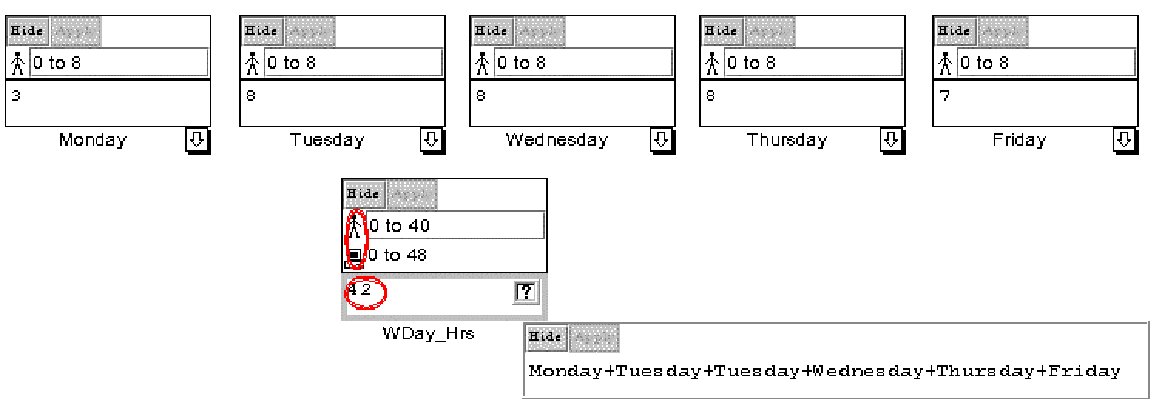

Figure 1: The Forms/3 environment. Cell formulas can be displayed via the tab at the lower right hand side of the cell, as has been done in WDay_Hrs. Additionally, cells with non-constant formulas have borders colored depicting "testedness" and a check box with a "?" in it, as shown in the right hand corner of the WDay_Hrs cell.

invoked again it might consider a different selection of values for Monday such as 1 and 2, which would widen the HMT assertion to "-1 to 2."

The primary job of the HMT assertions is to surprise the user, and thereby to generate user interest in "real" assertions (i.e., user-entered and computer-generated assertions). Thus, HMT assertions are—by design—usually bad guesses. The worse the guess, the bigger the surprise.

Consider Figure 1 and Figure 2, which are part of a weekly payroll program that is a running example in this paper. The user may expect values for Monday to range from 0 to 8, and rightly so, because employees cannot be credited with fewer than 0 or more than 8 hours per day. Since HMT was not aware of this, it attempted inputs less than zero. Thus, the HMT assertion for Monday probably violates the user's assumptions about the correct values for Monday. This is precisely what triggers curiosity according to the information-gap perspective.

Once an HMT assertion has been generated, it behaves as any assertion does. Not only does it propagate, but if a value arrives that violates it, the value is circled in red. This happens even as HMT is working to generate values. Thus, red circles sometimes appear as HMT is doing its transparent search for suitable test cases. These red circles are another use of surprise.

It is important to note that, although our strategy rests on surprise, it does not attempt to rearrange the user's work priorities by requiring users to do anything about the surprises. No dialog boxes are presented and there are no modes. HMT assertions are a passive feedback system; they try to win user attention but do not require it. Our behavior study revealed that users did not always attend to HMT assertions for the first several minutes of their task; thus it appears that the amount of visual activity is reasonable for requesting but not demanding attention.

Explanations: Encouraging Exploration

In our strategy a feature that surprises a user must inform the user. We have devised an on-demand explanation system structured around each object that might arouse curiosity. Users can begin exploring the object by viewing its explanation, on demand, in a low-cost way via tool tips.

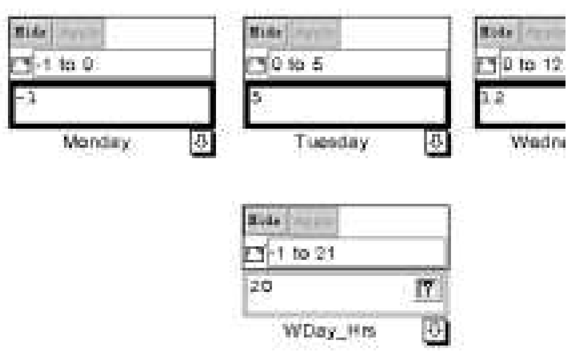

Figure 2: HMT has guessed assertions for the input cells (top row). (Since HMT changed the values of the input cells, they are highlighted with a thicker border.) The guesses then propagated through WDay_Hrs's formula to create an HMT assertion for that cell as well.

For example, when a user mouses over an HMT assertion they receive explanation 3 in Figure 3. The explanation describes the semantics: the computer was responsible for creating this assertion, and the assertion was a product of the computer's testing. The end of the explanation suggests a possible action for them to try (fixing the assertion) and specifies a potential reward (protecting against bad values).

Note that the computer "wonders" about this assertion. This makes explicit that the HMT assertion may not be correct and that the computer would like the user's advice. Previous empirical work has revealed that some users think the computer is always correct (e.g., [1]). Thus it is important to emphasize the tentative nature of the HMT assertions.

The explanation system spans all the objects in the environment. The principles governing these explanations are given in Table 1. In general, the three main components of explanations include: the semantics of the object, suggested action(s) if any, and the reward.

Including the semantics, action, and reward as part of the explanation are not arbitrary choices. Regarding semantics, although many help systems for end users focus mostly on syntax, a study assessing how end users learn to use spreadsheets found that the successful users focused more on the semantics of the spreadsheet than on syntax [19]. Regarding actions, Reimann and Neubert are among those who have examined learning by exploration. They point out that users (novice or experts) often form sub-goals using clues in the environment. The actions in the explanations suggest such sub-goals. Getting the user to take action in order to learn is, in fact, a principle of the minimalist model of instruction [21, 23]. Regarding reward, the attention investment model emphasizes the fact that the suggested action will cost the user effort and that, unless the potential rewards are made clear, the users may not be able to make an informed decision about whether or not to expend the effort.

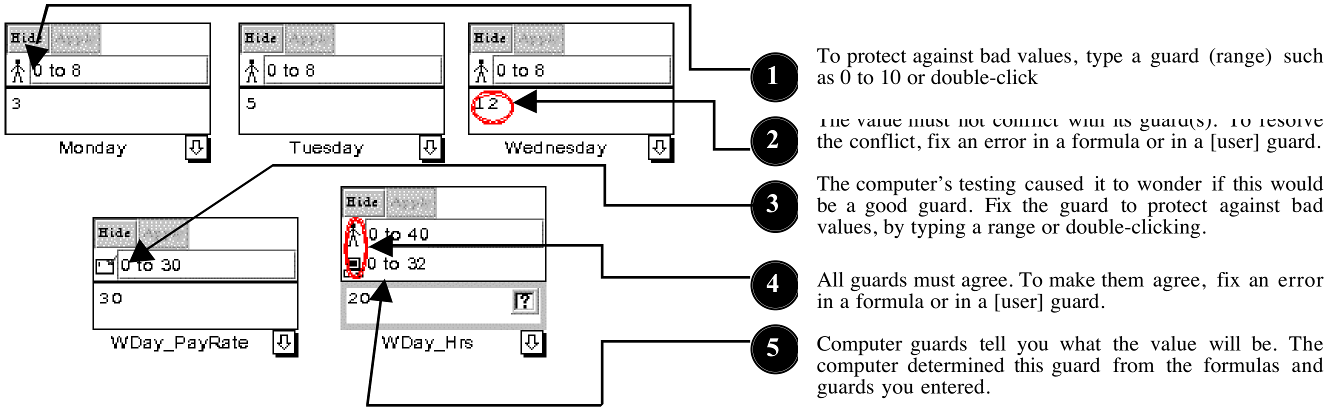

Figure 3: Five explanation examples.

Rewards

When the user edits an HMT assertion to create a user-entered assertion, there are three types of short-term rewards that can follow. There is also a fourth, longer term reward, namely the link to the surprise-explain-reward strategy as applied to "real" assertions.

The first reward, which visibly occurs in some situations, is input value validation. This reward can occur immediately when the user edits an HMT assertion. Consider again cell Monday in Figure 2. Suppose the user notices the HMT assertion, reads explanation 3 from Figure 3 and, deciding to take the explanation's advice to fix the assertion, changes it to "0 to 8." (The HMT assertion helps show how to succeed by acting as a template, exemplifying assertion syntax.) Despite the assertion entry, the cell's *value* is still –1, and the system circles the value, since it is now in violation with its assertion. Thus, by taking the advice of the system the user has been immediately rewarded: the system is indeed "protect(ing) against bad values."

The second reward always occurs. Once a user places an assertion on an input cell, the behavior of HMT changes to honor the assertion. Continuing the previous example, once cell Monday has the assertion "0 to 8" and the user runs HMT again, HMT will always choose values satisfying the assertion. Since HMT's "thought process" is displayed as it mulls over values to choose, this behavior change is noticeable to some users, as the section about our study will discuss. Since HMT's selected values can seem odd from the user's perspective, given their knowledge of the program's purpose, getting HMT to choose sensible input values is rewarding if noticed.

Device type	Explanation
Simple interaction device.	Purpose, e.g., "Close Window."
Interaction device for a software engineering task.	Action, reason, semantics. E.g., explanation 1 in Figure 3.
Feedback device for a software engineering task.	Semantics. If there is a suitable follow-up action, suggest it and give the reason. E.g., explanation 4 in Figure 3.
A device about which there is nothing to add beyond its label.	Nothing. (Principle: no worth-less explanations.) E.g., a clearly labeled button.

Table 1: The explanation system's guiding principles.

The third reward also pertains to changes in HMT's behavior. HMT becomes an aggressive seeker of test values that will expose bugs. As other test generators in the software engineering community have done (e.g., [11]), HMT attempts to violate user-entered or computer-generated assertions. (This behavior is not yet implemented in the current prototype.) This behavior is focused on non-input cells (i.e., cells that have formulas instead of constant values), and potentially creates a value violation. A value violation on a non-constant cell indicates one of three things: a faulty assertion, a situation in which the program could fail given inappropriate values in input cells not protected by assertions, or an outright bug. For example, for cell WDay_Hrs, HMT will attempt to violate the assertion "0 to 40" by looking for values in the inputs contributing to WDay_Hrs that produce a value violation. When HMT's pursuit of bugs succeeds, the user is not only rewarded, but also surprised again, leading to a longer term use of the surprise-explain-reward strategy.

LONG TERM SURPRISE-EXPLAIN-REWARD

HMT assertions are intended to help users learn and appreciate assertions, but after that goal has been accomplished, some users will not need HMT's guessed assertions. They will have learned how to enter assertions, regardless of whether HMT guesses assertions on the cells they wish to protect. The surprise-explain-reward strategy carries over to a longer term, to help maintain correctness.

A bug is almost always a surprise. Thus, if users are to find bugs, they need to be surprised. Our strategy provides bug-related surprises using the user-entered assertions. These surprises are themselves rewards, since they mean a bug has been semi-automatically identified.

Assertions can lead to three kinds of surprises. First, the value violations are surprises. As already discussed, they identify bugs or assertion errors, and are circled in red. Second, assertion conflicts are surprises. As explained earlier, assertion conflicts arise when the system's propagating the user-entered assertions through formulas produce computer-generated assertions that disagree with user-entered assertions. Like value violations, they are circled in red (as in WDay_Hrs in Figure 1) and indicate bugs or assertion errors. Third, the computer-generated assertions might "look wrong" to the user. The explanations in Figure 3 support all three surprises by explaining their semantics and suggesting suitable actions.

BEHAVIOR STUDY

To investigate how the surprise-explain-reward strategy would affect users' debugging behavior, we conducted a behavior study with 16 participants. The study replicated most aspects of the experiment reported in [4]. In both studies, the same problems were used, and WYSIWYT and Help-Me-Test were explained to the participants. In the earlier experiment, assertions were explained and were also automatically provided, on demand, to participants, and the surprise-explain-reward strategy was not present. In contrast, in the study we describe here, assertions were not provided, and the surprise-explain-reward strategy was in place. Most important, we did not present assertions—in fact, they were never even mentioned. Instead, participants were simply introduced to the mouse-overs and given time to explore.

We asked the participants to test the two programs to make sure they worked according to their description, and that if they found any errors they should fix them. One program was the "weekly pay" example we have used as a running example in this paper, shown in its entirety in Figure 4. The other program was the Grades program described in [4]. Time limits were imposed to ensure working on both problems. The order of the problems was counterbalanced.

Success Introducing Assertions

Was the strategy successful in enticing users to create and interact with assertions at least once?

Because this question depends on sequence of actions, our data are presented from the perspective of sequence, i.e., Task 1 or Task 2, irrespective of which program was first for that participant.

Of the 16 participants, 15 (94%) of them did choose to enter assertions in at least one task, and 14 (87%) used assertions during both tasks. Recall that assertions were never mentioned or implied in the tutorial, so the reason for this activity can only be either explanations alone or HMT assertions along with its related explanations.

But, which one was the reason? As Table 2 shows, in the first task, 74% of the assertions resulted from directly modifying HMT assertions. This indicates that HMT assertions were the way most participants were enticed to begin interacting with assertions. By the second task, participants had learned enough to enter assertions without HMT assertions, and the number of assertions entered by editing HMT assertions dropped to 33%. However, there was no drop in the total number of assertions, which shows that interest in assertions did not decline after the first task; only the users' need for HMT's help with assertion entry declined.

Explain/Explore

Did users seek explanations of surprising objects?

The explanation system bore the primary responsibility for educating the participants about assertions, once the surprises got their attention. To determine whether or not participants actively used the explanations, we recorded the participants' accesses to the explanations.

The participants actively sought explanations: more than 2800 explanations were recorded during the experiment. Interest in assertions, including HMT assertions, was clear: 12% of the explanations displayed were related to assertions.

Were the explanations important in tying together the surprises with the rewards for HMT assertions?

To answer this question, we examined the data to see if participants followed the following pattern: the user is surprised, the user seeks an explanation for the subject of the surprise, the user takes an action specified by a system explanation.

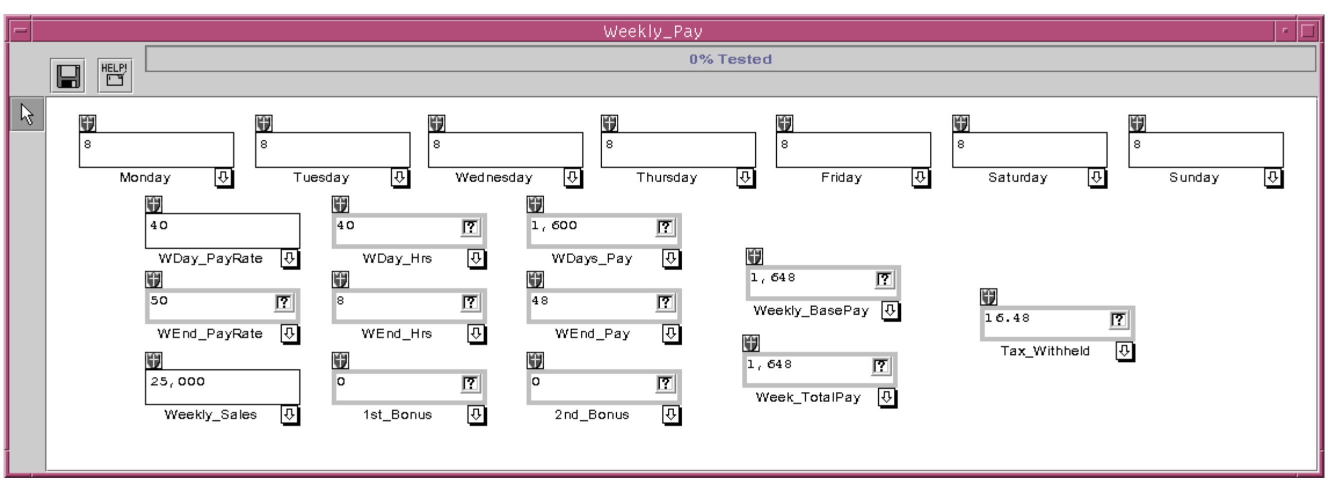

Figure 4: One of the programs used in the behavior study.

Assertions produced	Using HMT assertions	Not using HMT assertions
Task 1	74%	26%
Task 2	33%	67%

Table 2: How users entered assertions.

We were surprised at the large extent to which this pattern existed for HMT assertions. Fully 56% of participants followed the pattern when interacting with HMT assertions. A related pattern without explanations was also present: after HMT assertions appeared, some participants entered a user assertion without looking for the explanation. Of all participants, 94% followed either the first or the second pattern described above.

Sufficiency of the Rewards

Were the rewards sufficient to encourage users to continue using the assertions feature?

Our study provided two types of evidence that the rewards were indeed sufficient for this. The strongest and most general is the fact that, once participants used assertions once, they used them again. We have already provided some evidence of this. Additional evidence is that, although participants did not enter assertions until almost 14 minutes (mean) into the first task, by the second task they began entering assertions much earlier (see Table 3). In fact, 9 of the 16 users (56%) entered assertions within the first minute of the second task. It seems clear that after users became familiar with assertions during the first task, they had learned to recognize their value by the second task.

The other type of evidence comes from comments on the post-session questionnaire. One user's comment pertained to the long-term reward: "I was trying to figure out what they did, then I found them reassuring because I think they help with accuracy." Another user described the reward of controlling HMT's behavior: "I did so to try and get the help feature to stay within my limits."

Correctness

Ultimately, were users successful in creating accurate assertions?

The participants created 279 assertions during the study, and 95% of them were correct. By "correct," we mean identical to assertions that had previously been devised by the design team of the earlier experiment, in which assertions had been automatically provided for many of the cells.

Did the users' assertions help with program correctness?

We already know from [4] that participants using assertions provided to them—debugging the same programs as in this study—corrected significantly more bugs than participants not using assertions. What we now consider is whether assertions created by the participants themselves still contributed to correctness.

Since each program had different bugs, we report the information program by program rather than task by task. A value violation or assertion conflict was credited with helping the user fix a bug if the display of the red oval was followed by one or more formula edits and the formula edits resulted in a corrected formula. The results of using these criteria on the transcripts can be seen in Table 4. Overall, 46% of the value violation ovals and 56% of the assertion conflicts pointed out bugs that were eventually fixed. Since the problems had a time limit, no participants were able to reach 100%.

	Average time to enter first assertion
Task 1	13:26 minutes
Task 2	3:40 minutes

Table 3: Time until entry of first assertion.

Obviously, the assertion feedback was tied to some bug fixes. A critical question is to what extent the participants believed the programs were now correct. In essence, the strategy's purpose is to maintain users' curiosity in their program's correctness "long enough" to get all the bugs out. Thus, we compared their correctness scores with their self-ratings of the programs' correctness at the end of the tasks. The Spearman Rank Correlation (Table 5) indicates that the participants were able to accurately judge their debugging progress.

Program	Value violations: fixes / circles	Assertion conflicts: fixes / circles
Weekly Pay	3 / 10	10 / 11
Grades	15 / 29	20 / 43

Table 4: Number of bugs fixed after a red circle pointed it out.

Program	Rho	P-value
Weekly Pay	.657	.0109
Grades	.524	.0423

Table 5: Correlations between participants' self-ratings and actual correctness.

CONCLUDING REMARKS

We have presented a curiosity-centered approach for enticing end users to enter and use assertions in end-user programming. Our approach attempts to first arouse users' curiosity and then to encourage them to follow through, using a strategy comprised of an intertwined collection of surprises, explanations, and rewards.

We evaluated the strategy by examining the behavior of 16 participants. Although they received no instruction about assertions and had to learn about them through exploration, almost all participants chose to enter at least one assertion. The primary vehicle they used to get started with assertions was HMT assertions, which shows that HMT's surprises did arouse their curiosity. Once they discovered assertions, most participants chose to enter quite a few of them (averaging 18 per participant), which clearly shows follow-through. The data show that the follow-through was due both to the explanations and to the rewards. Finally, the data from our study, especially when considered in light of results from a related study, provide clear evidence of the assertions' positive impacts on correctness.

Our next step will extend the surprise-explain-reward strategy to other end-user software engineering devices, so as to further harness curiosity to help users increase the correctness of their programs.

ACKNOWLEDGMENTS

We thank Marc Fisher for his help with HMT. This work was supported in part by NSF under ITR-0082265. Also, Ledah Casburn was supported in part by the Computing Research Association's Distributed Mentor Project, which is funded in part by NSF under EIA-0124641.

REFERENCES

1. Beckwith, L., Burnett, M., Cook, C. Reasoning about many-to-many requirement relationships in spreadsheets, in *Proc. IEEE Symp. Human-Centric Computing*, (Arlington VA, Sept. 2002), IEEE, 149-157.
2. Blackwell, A. and Green, T. R. G. Investment of attention as an analytic approach to cognitive dimensions. In T. Green, R. Abdullah & P. Brna (Eds.) *Collected Papers Wkshp. Psychology of Programming Interest Group*, 1999, 24-35. www.cl.cam.ac.uk/~afb21/publications/PPIG99.htm
3. Boehm, B., Abts, C., A. Brown, W., Chulani, S., Clark, B., Horowitz, E., Madachy, R., Reifer, J., and Steece, B. *Software Cost Estimation with COCOMO II*. Prentice Hall PTR, Upper Saddle River, NJ, 2000.
4. Burnett, M., Cook, C., Pendse, O., Rothermel, G., and Summet, J. End-user software engineering with assertions, Technical Report 02-60-05, Oregon State University, Sept. 2002. ftp://ftp.cs.orst.edu/pub/burnett/TR.026005-assertions.pdf
5. Burnett, M., Atwood, J., Djang, R., Gottfried, H., Reichwein, J., Yang, S. Forms/3: a first-order visual language to explore the boundaries of the spreadsheet paradigm. *J. Functional Programming 11*, 2, (March 2001) 155-206.
6. DiGiano, C., Kahn, K., Cypher, A., and Smith, D.C. Integrating learning supports into the design of visual programming systems. *J. Visual Languages and Computing 12*, 5 (October 2001), 501-124.
7. Fisher, M., Cao, M., Rothermel, G., Cook, C., Burnett, M. Automated test generation for spreadsheets, in *Proc. ICSE '02*, (Orlando FL, May 2002), IEEE 141-151.
8. Green, T. R. G. and Petre, M. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *J. Visual Languages and Computing 7*, 2 (June 1996), 131-174.
9. Hastie, R. Causes and effects of causal attribution. *J. Personality and Social Psychology, 46*, (1984), 44-56.
10. Jussien, N. and Ouis, S., User-friendly explanations for constraint programming, in *Proc. ICLP'01 11th Wkshp. Logic Programming Environments* (Paphos Cyprus, Dec 2001).
11. Korel, B. and Al-Yami, A.. Assertion-oriented automated test data generation, in *Proc. ICSE '96* (Berlin Germany, March 1996), IEEE, 71-80.
12. Krishna, V., Cook, C., Keller, D., Cantrell, J., Wallace, C., Burnett, M., and Rothermel, G. Incorporating incremental validation and impact analysis into spreadsheet maintenance: an empirical study, in *Proc. Software Maintenance*, (Florence Italy, Nov. 2001), IEEE, 72-81.
13. Lieberman, H. Interfaces that give and take advice. *Human-Computer Interaction for the New Millenium*, John Carroll, ed., ACM Press/Addison-Wesley, 475-485, 2001.
14. Lieberman, H. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, San Francisco, 2001.
15. Lowenstein, G. The psychology of curiosity. *Psychological Bulletin 116*, 1 (1994), 75-98.
16. Miller, R. and Myers B. Outlier finding: focusing user attention on possible errors, in *Proc. User Interface Software and Technology*, (Orlando FL, Nov. 2001), ACM Press, 81-90.
17. Nardi, B. *A Small Matter of Programming: Perspectives on End-User Computing*, MIT Press, Cambridge, MA, 1993.
18. Panko, R. What we know about spreadsheet errors. *J. End User Computing*, (Spring 1998). panko.cba.hawaii.edu/ssr.
19. Reimann, P. and Neubert, C., The role of self-explanation in learning to use a spreadsheet through examples. *J. Computer Assisted Learning 16*, (Dec. 2000), 316-325.
20. Rosenblum, D. A practical approach to programming with assertions. *IEEE Trans. Soft. Eng. 21,* 1 (Jan. 1995), 19-31.
21. Rosson, M. and Seals, C. Teachers as simulation programmers: minimalist learning and reuse, in *Proc. CHI '01,* (Seattle, WA, Apr. 2001), ACM, 237-244.
22. Rothermel, G., Li, L., DuPuis, C. and Burnett, M. What you see is what you test: a methodology for testing form-based visual programs, in *Proc. ICSE '98*, (Kyoto Japan, Apr. 1998), IEEE, 198-207.
23. Seals, C., Rosson, M., Carroll, J., Lewis, T., and Colson, L. Fun learning stagecast creator: an exercise in minimalism and collaboration, in *Proc. IEEE Symp. on Human-Centric Computing 2002* (Arlington VA, Sept. 2002), IEEE, 177-186.
24. Wilcox, E., Atwood, J., Burnett, M., Cadiz, J., and Cook, C. Does continuous visual feedback aid debugging in direct-manipulation programming languages? In *Proc. CHI '97*, (Atlanta GA, March 1997), ACM Press, 258-265.